Hochschule der Medien Stuttgart

Implementierung eines LISP Interpreters in Smalltalk

Verfasser:	Benjamin Zaiser
E-Mail:	bz003@hdm-stuttgart.de
Studiengang:	Computer Science and Media
Seminar:	Design und Implementierung fortgeschrittener Programmiersprachen (38360)
Semester:	3
Datum:	1. Juli 2009

1 Aufgabenstellung

Es soll ein LISP-Interpreter in der Programmiersprache Smalltalk implementiert werden. Dabei soll auf keinen bereits vorhandenen Quellcode aufgebaut werden, sondern sozusagen „from scratch“ begonnen werden. Als Smalltalk-Umgebung kommt dabei Cincom VisualWorks 7.6 zum Einsatz. Das Ziel war dabei, einen funktionsfähigen Interpreter zu implementieren, ohne dabei Rücksicht auf die Performance zu nehmen.

Bei der Entwicklung wurde auf das Paper „The Roots of Lisp“ (1) von Paul Graham zurückgegriffen. In dem Paper werden die Konzepte, die ursprünglich von John McCarthy entwickelt wurden, relativ gut erklärt. Dieses Paper bildet die Ausgangsbasis und den „roten Faden“ bei der Entwicklung des Interpreters.

2 Der Quellcode

Der Smalltalk-Quellcode ist in der Datei MyLISP.st zu finden. Dieser kann in jede Smalltalk-Umgebung (z.B. VisualWorks, Smalltalk/X, Dolphin Smalltalk, GNU Smalltalk) eingebunden und ausgeführt werden. Hier eine kleine Anleitung, wie der Quellcode in VisualWorks eingebunden werden kann:

1. VisualWorks starten
2. Klassenbrowser öffnen
3. Ein neues Package anlegen, z.B. „MyLISP“
4. Rechtsklick auf das neue Package
5. „FileInto“ anklicken
6. Die Datei „MyLISP.st“ öffnen

Als nächstes muss das Parcel „Regex11“ in das System geladen werden, da der Interpreter Regular-Expressions verwendet. Unter VisualWorks kann das Parcel wie folgt geladen werden:

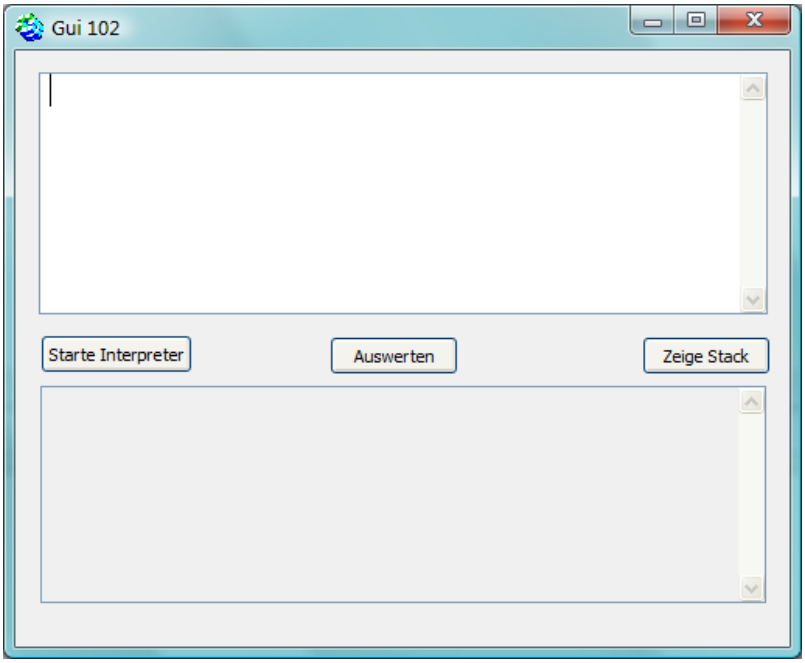
1. Im Launcher-Fenster klick auf Parcel-Manager
2. Klick auf „Advanced Utilities“
3. Rechtsklick auf das Parcel „Regex11“
4. Klick auf „Load“

Jetzt sind alle Klassen und benötigte Bibliotheken im System geladen. Um nun die Applikation zu starten (bzw. das Interpreter-Fenster zu öffnen) müssen folgende Befehle ausgeführt werden:

1. Klick auf Klasse „InterpreterGUI“
2. Klick auf Reiter „Class“
3. Klick auf Klassenmethode „windowSpec“
4. Unter Reiter „Visual“, klick auf „Open“

Es kann auch einfach das Image MyLISP.im in Cincom VisualWorks 7.6 geladen werden (`http://www.benjamin-zaiser.de/uploads/media/MyLISP_Image.zip`)

3 Die Bedienung



Zunächst muss der Interpreter durch Klick auf „Starte Interpreter“ gestartet werden. Im oberen Textfeld kann der LISP Code eingegeben werden. Es kann dabei immer nur ein Befehl eingegeben werden, ähnlich der Read-Eval-Print Schleife. Um den Befehl auszuwerten, muss der Button „Auswerten“ angeklickt werden. Das Ergebnis der Auswertung erscheint im unteren Textfeld. Es ist auch möglich, eigene Funktionen zu deklarieren, die dann auf dem Stack abgelegt werden. Um nachschauen zu können, welche Funktionen sich aktuell auf dem Stack befinden, kann der Button „Zeige Stack“ angeklickt werden. Es werden dann der Funktionsname und die Funktionsdefinition im unteren Textfeld ausgegeben. Um den Stack zu leeren muss der Interpreter neu gestartet werden. Hierfür einfach den Button „Starte Interpreter“ anklicken.

4 Mögliche Eingaben

Es stehen die Basisfunktionen wie z.B. „atom“, „equal“, „car“, „cdr“, „cons“, „cond“, etc. zur Verfügung. Ebenfalls wurden anonyme Funktionen (Lambda) implementiert. Einer Lambda-Funktion kann durch den Befehl „label“ ein Name zugewiesen werden. Die Funktion wird daraufhin auf dem Stack abgelegt und steht dann zum späteren Aufruf bereit. Eine Liste aller möglichen getesteten Funktionen ist im Anhang gelistet.

5 Funktionsweise

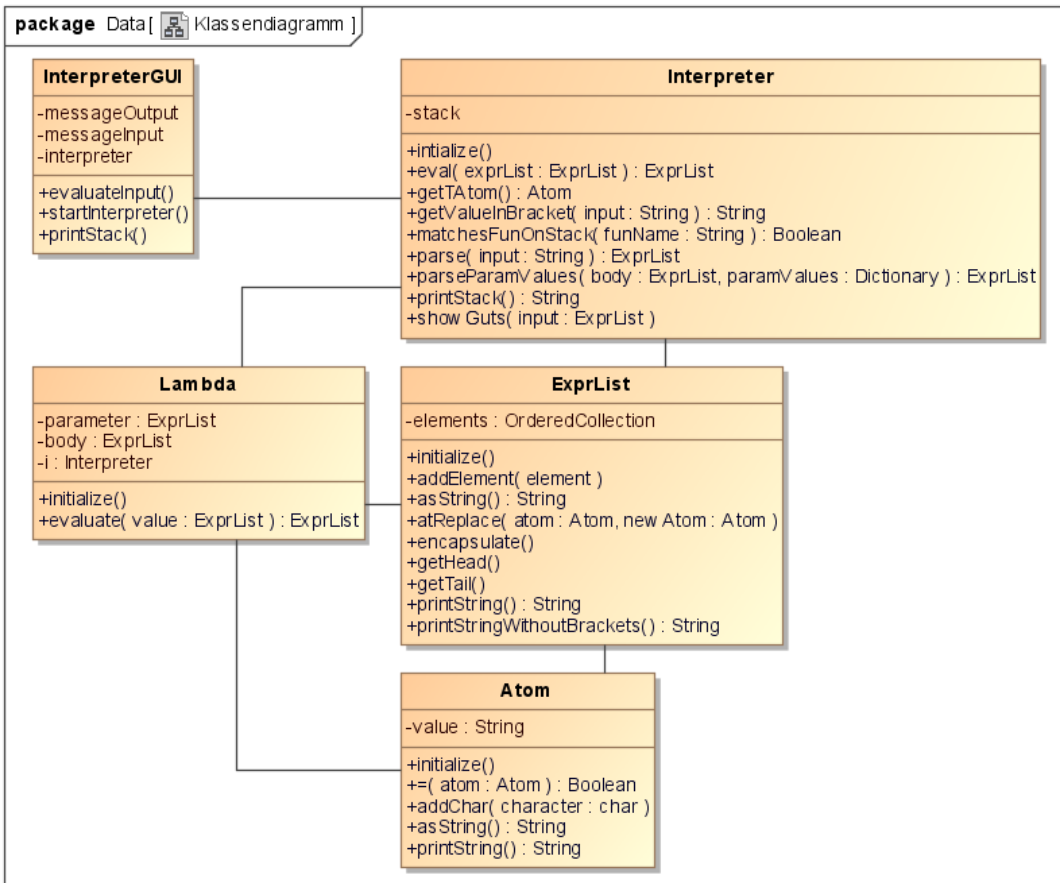


Abbildung 1: Klassendiagramm

Die „Main“-Methode entspricht der evaluateInput() in der Klasse InterpreterGUI. Sie ruft die Methode „parse“ und „eval“ der Interpreter Klasse auf und gibt das Ergebnis im unteren Textfeld aus.

Die Methode parse der Interpreter Klasse erstellt aus dem Eingabe-String eine ExprList, die wiederrum Atom- oder weitere ExprList-Objekte enthalten kann (dies entspricht sozusagen dem Abstract Syntax Tree). Die Parse-Methode funktioniert wie folgt:

1. Setze Pointer auf 0.
2. Erzeuge neue ExprList.
3. Lese das erste Zeichen.
4. Wenn erstes Zeichen = „(“, dann kopiere den gesamten Inhalt des Strings bis zum zugehörigen) -Zeichen und rufe die parse Methode rekursiv auf. Das Ergebnis der Parse-Methode wird der ExprList angehängt und der Pointer wird um die entsprechende Anzahl der Zeichen innerhalb der Klammern erhöht. [1]
5. Wenn erstes Zeichen = „Zahl oder Ziffer“, dann erzeuge neues Atom-Objekt und hänge das Zeichen an das Atom-Objekt an.
6. Wenn erstes Zeichen = „Leerzeichen“, dann hänge das Atom-Objekt an die ExprList an.
7. Inkrementiere den Pointer.
8. Gebe die ExprList zurück.

Die Methode „eval“ der Interpreter Klasse bekommt die ExprList, die von parse erstellt wurde und wertet sie entsprechend aus.

1. Wenn die Eingabe ein Atom ist, setze result auf Eingabe, sonst lese das erste Atom der ExprList aus.
2. Wenn erstes Atom = „Zahl oder Ziffer“, result = erstes Atom.
3. Wenn erstes Atom = „quote oder q“, dann setze result auf das zweite Element der ExprList.
4. Wenn erstes Atom = „eq“, ...
5. ...
6. Wenn erstes Atom = „lambda“, dann erzeuge neues Lambda Objekt, setze Parameterliste und „Methodenrumpf“ und setze result auf erzeugtes Lambda.
7. Wenn erstes Atom = ExprList, dann rufe Methode evaluate des Lambda Objekts mit den Parameter-Werten auf und setze result auf das Ergebnis. (Das Lambda-Objekt ist zu diesem Zeitpunkt durch die vorhergehende Regel bereits erstellt worden.)

[1] Hier gibt es noch Verbesserungspotential. Durch die Verwendung eines Streams (von Smalltalk) könnte der Pointer entfernt werden. Dadurch müsste beim rekursiven Aufruf der String innerhalb der Klammer nicht extra kopiert werden.

Die Methode „evaluate“ des Lambda-Objekts ersetzt dabei alle Vorkommnisse der definierten Parameter im Methodenrumpf mit dem entsprechenden Parameter-Wert. Es wird hierbei einfach auf dem LISP Befehls-String operiert. Danach wird der String erneut geparsed und evaluiert. Dieser Vorgang hat den Nachteil, dass innerhalb des Methodenrumpfs kein weiteres Lambda definiert werden darf, da ansonsten die Parameter des inneren Lambdas mit den Parameter-Werten des ersten Lambdas ersetzt werden. Die Funktion, die die Parameter durch die Werte ersetzt, müsste einfach das innere Lambda ignorieren.

6 Fazit

Die Entwicklung eines LISP-Interpreters beginnend bei null war ein sehr hochgestecktes Ziel. Die Funktionen der „parse“ und „eval“ Methode sind bereits durch die vielen rekursiven Aufrufe relativ komplex. Hinzu kommen einige Probleme mit der Sprache Smalltalk und der Entwicklungsumgebung an sich, da ich zuvor noch nie mit Smalltalk gearbeitet habe. Dementsprechend hoch war dann aber auch der Lernerfolg. Es ist faszinierend, wie durch die Rekursion sich Probleme ab einem bestimmten Punkt „in Luft auflösen“. Dies sieht man besonders schön auf der Konsole, während der Evaluation von (pair (q (x y z)) (q (a b c))).

Literatur

[1] GRAHAM, Paul: The Roots of Lisp. 2002

Anhang

Diese Funktionen orientieren sich an dem Paper von Paul Graham und wurden entsprechend getestet (alles, nach „->“, entspricht der Ausgabe):

```
(q a)
-> a
```

```
(atom (q a))
-> t

(atom (q (a b c)))
-> ()
```

```
(eq (q a) (q a))
-> t

(eq (q a) (q b)))
-> ()
```

```
(car (q (a b c)))
-> a

(cdr (q (a b c)))
-> (b c)
```

```
(cons (q a) (q b)))
-> (a b)

(cons (q a) (q (b c))))
-> (a b c)

(cons (q a) (cons (q b) (cons (q c) (q ()))))))))
-> (a b c)

(car (cons (q a) (q (b c)))))
-> a

(cdr (cons (q a) (q (b c)))))
-> (b c)
```

```
(cond ((eq (q a) (q b)) (q first))
      ((atom (q a)) (q second)))
-> second
```

```
((lambda (x) (cons x (q b))) (q a))
-> (a b)

((lambda (x) (cons x (q (b c)))) (q a))
-> (a b c)

((lambda (x y) (cons x (cdr y))) (q z) (q (a b c)))
-> (z b c)

((lambda (f) (f  (q (b c)))) (lambda (x) (cons (q a) x)))
-> (a b c)
```

```
(label test (lambda (x) (cons x (q b))))
-> (test (q a))
```

```
(label null
  (lambda (x)
    (eq x (q ()))
  )
)

(null (q a))
-> ()
(null (q ()))
-> t
```

```
(label and (lambda (x y)
  (cond (x (cond (y (q t)) ((q t) (q ())))
         ((q t) (q ())))
)))

(and (atom (q a)) (eq (q a) (q a)))
-> t
(and (atom (q a)) (eq (q a) (q b)))
-> ()
```

```
(label not
  (lambda (x)
    (cond (x (q ()))
          ((q t) (q t)))
  )
)

(not (eq (q a) (q a)))
-> ()
(not (eq (q a) (q b)))
-> t
(eq (not (atom (q a))) (not (atom (q a))) )
-> t
```

```
(define list2 (lambda (a b) (cons a (cons b (q ())))))
(list2 1 2)
-> (1 2)

(define list3 (lambda (a b c) (cons a (cons b (cons c (q ()))))))
```

```
(define pair
  (lambda (x y)
    (cond ((and (null x) (null y)) (q ()))
          ((and (not (atom x)) (not (atom y)))
           (cons (list2 (car x) (car y))
                 (pair (cdr x) (cdr y))))
    )
  )
)
(pair (q (x y z)) (q (a b c)))
-> ((x a) (y b) (z c))
```